CUDA-based multi-core implementation of MDS-based bioinformatics algorithms

Thilo Fester
Martin-Luther-University Halle-Wittenberg
thilo.fester@student.uni-halle.de

Falk Schreiber
IPK Gatersleben & Martin-Luther-University Halle-Wittenberg
schreibe@ipk-gatersleben.de

Marc Strickert
IPK Gatersleben
stricker@ipk-gatersleben.de

Abstract: Solving problems in bioinformatics often needs extensive computational power. Current trends in processor architecture, especially massive multi-core processors for graphic cards, combine a large number of cores into a single chip to improve the overall performance. The Compute Unified Device Architecture (CUDA) provides programming interfaces to make full use of the computing power of graphics processing units. We present a way to use CUDA for substantial performance improvement of methods based on multi-dimensional scaling (MDS). The suitability of the CUDA architecture as a high-performance computing platform is studied by adapting a MDS algorithm on specific hardware properties. We show how typical bioinformatics problems related to dimension reduction and network layout benefit from the multi-core implementation of the MDS algorithm. CUDA-based methods are introduced and compared to standard solutions, demonstrating 50-fold acceleration and above.

1 Introduction

Bioinformatics is faced with accelerating increase of data set sizes originating from powerful high-throughput measuring devices. The implementation of computational intensive tasks in parallel technology is one of the key solutions to time-efficient data processing. Today, often compute jobs are performed on cluster computers or on large multi-core servers to take advance of parallelization. We will discuss an evolving path to provide work-efficient, parallel and desktop-suitable solutions based on acceleration by graphics processing units using the compute unified device architecture (CUDA) for computation on commonly available graphics processing units (GPU). High-throughput multi-dimensional scaling (HiT-MDS) is a versatile tool for biological data analyses that is systematically transferred to the GPU for taking advantages of the massively parallel hardware architecture for scientific computing.

1.1 Multidimensional Scaling

Multidimensional scaling (MDS) is a data processing method suitable for addressing several analytical purposes: (i) for dimension reduction of vector data, providing a nonlinear alternative to the projection to principal components; (ii) for the reconstruction of a data dissimilarity matrix of pairwise relationships in the Euclidean output space; (iii) for conversion of a given metric space, such as data compared by Manhattan distance, into Euclidean space, (iv) for dealing with missing data relationships using zero force assumption. These features make MDS a valuable tool for the analysis of large data tables and for dealing with (partial) information about data relationships [IMO09, SSUS07, TO05]. We focus on two examples of MDS application: one is related to dimension reduction in gene expression time series data, the other one is related to network layout from adjacency information.

1.2 GPGPU Programming with NVIDIA CUDA

In the last ten years general purpose computing on graphics processors became more and more important. Higher memory bandwidth, increasing (parallel) floating point performance compared to CPUs and rising memory capacities as well as low costs get attractive to scientists because of impressive speed up factors of up to several hundred times in different CUDA based analyses [BK09, GHGC09, JK09, LKPM09]. Following the trend to take advance of a little 'supercomputer at home', approaches with massive parallelism on the GPU have been implemented in scientifically important tools such as MATLAB or FORTRAN libraries [FJ07, GDD08].

Because of the development from simple graphics devices into highly parallel, multithreaded many-core processors, today GPUs are very appropriate to solve problems transformable into data parallel instructions operation. That is, the more independent subsequent instructions are the lower is the communication overhead which usually causes performance loss. By massive parallel operations memory access latency can even be avoided by in-place recalculations instead of accessing big data caches. For that purpose, parallel instructions are embedded into a logical grid of thread blocks, which is mapped to scalar processors by the instruction unit of a multiprocessor as illustrated in Figure 1. This architecture is called SIMT (single-instruction, multiple-thread) which is similar to the well-known SIMD (single-instruction, multiple-data) concept [Cor08].

For controlling the GPU computation CUDA was developed as a hybrid CPU-GPU interaction model. The above mentioned single-instruction functions are called from a CPU thread (referred to as host in the following). Such functions are called kernels whose instructions and amount of executed threads can be specified by the coder. As shown by Ryoo et al. not only aiming at best local acceleration, but also the distribution of threads within the grids and blocks can significantly influence the performance [RRS$^+$07].

Another important feature of NVIDIA's CUDA enabled devices is the heterogeneous memory (see Figure 1). The large global memory reaching gigabytes of capacity contains small

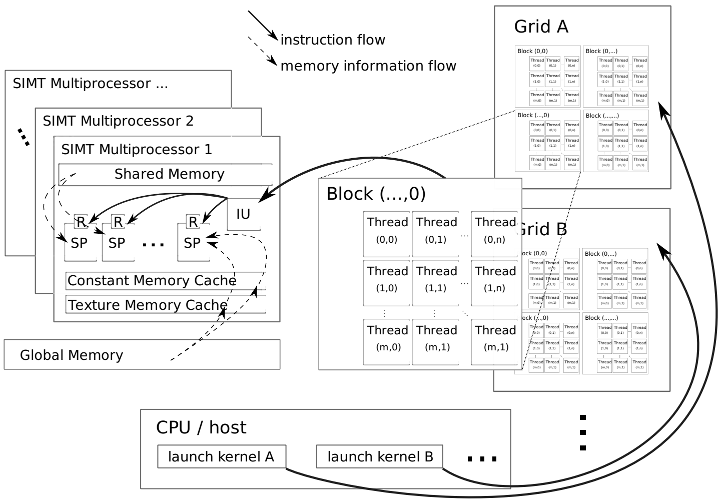

Figure 1: CUDA memory, processor and programming model. IU - Instruction Unit, R - Register, SP - Scalar processor

subsets of two cached memory types, texture and constant memory, which are available in every thread and accessible as fast as registers after being cached. The on-chip shared memory is available within all threads of a block. In case of no bank conflicts, it is as fast as register. Bank conflicts occur if two threads try to read the same memory contemporaneously, which is then serialised [Cor08]. Nearly all implementations use these fast memories to achieve communication between threads. This leads to massive performance gain compared to the usage of global memory or the even much worse communication delay via host control [CMS08, Cor08, Sel08]. To get the best out of global memory Seland pointed out to access contiguous (coalesced) memory and reported speed up factors of up to ten by using this technique [Sel08].

Furthermore NVIDIA declares CUDA as an extension to the C programming language and targets to simplify parallel computation. Hence working with CUDA is quite intuitive, and there is a low learning curve even without much knowledge about graphics hardware or OpenGL. This makes CUDA attractive for tool development for bioinformatics tasks.

2 Methods

This section is organised as follows: First we describe the multi-dimensional scaling (MDS) method. Then the implementation and optimisation of the algorithms in CUDA are described in detail. We close this section with two important applications of MDS: gene expression analysis and automatic layout of biological networks.

2.1 Multi Dimensional Scaling

Very intuitive visualisation of relationships between different data records can be obtained by reconstructing these relationships as pairwise distances in the usual Euclidean 2D plane or 3D space. Usually data projections to the principal components are used for that purpose, referred to as PCA projection. However, PCA is restricted to linear mappings of high-dimensional data, thereby focusing on directions of maximum Euclidean variance. A more natural goal is to obtain a low-dimensional display of a Euclidean space that reflects most faithfully the similarities among the source data.

In principle, this goal can be reached by using multi-dimensional scaling (MDS) techniques. In classical approaches, distances between the reconstructed low-dimensional points should be maximum similar to distances between the original data records. This strict optimisation task can be very hard, though, because of ambiguous compromise solutions for complex source relationships being rendered into a low-dimensional target space. Most MDS methods define quite stringent cost functions, such as least squares approaches targeting identity of the distances between the reconstructed point locations and the distances of corresponding input data.

Alternatively, Pearson correlation $\mathsf{r} \in [-1;1]$ can be computed between the distance matrices $\mathbf{D} = (\mathrm{d}_{ij})_{i,j=1\ldots n}$ and $\hat{\mathbf{D}} = (\hat{\mathrm{d}}_{ij})_{i,j=1\ldots n}$ of input data and of reconstructed points, respectively, by

$$\begin{aligned} \mathsf{r}(\mathbf{D},\hat{\mathbf{D}}) &= \frac{\sum_{i<j}^{n} (\mathrm{d}_{ij} - \mu_{\mathbf{D}}) \cdot (\hat{\mathrm{d}}_{ij} - \mu_{\hat{\mathbf{D}}})}{\sqrt{\sum_{i<j}^{n} (\mathrm{d}_{ij} - \mu_{\mathbf{D}})^2} \cdot \sqrt{\sum_{i<j}^{n} (\hat{\mathrm{d}}_{ij} - \mu_{\hat{\mathbf{D}}})^2}} =: \frac{\mathscr{B}}{\sqrt{\mathscr{C} \cdot \mathscr{D}}} \\ \text{with} \quad \mu_{\dot{\mathbf{D}}} &= \frac{2}{n \cdot (n-1)} \cdot \sum_{i<j}^{n} \dot{\mathrm{d}}_{ij}\,, \quad \dot{\mathbf{D}} \in \{\mathbf{D},\hat{\mathbf{D}}\}\,, \dot{\mathrm{d}}_{ij} \in \{\mathrm{d}_{ij},\hat{\mathrm{d}}_{ij}\}\,. \end{aligned} \tag{1}$$

This correlation approach allows infinitely many more solutions than strict identity optimisation, while ensuring maximum correlation between source and target distances. Relaxation of the optimisation procedure is explained by the invariance of Pearson correlation against rescaling of vectors by a factor and against baseline shifts by an additive offset. The following method, called high-throughput multidimensional scaling (HiT-MDS), describes how correlation is used to help alleviate the optimisation task of finding proper low-dimensional point locations.

Referring to source vectors $\mathbf{x}^i \in \mathbf{X}$, target vectors $\hat{\mathbf{x}}^i \in \hat{\mathbf{X}}$ and their respective dimensions q and $\hat{q}$, the correlation $\mathsf{r}(\mathbf{D},\hat{\mathbf{D}})$ between entries of the source distance matrix $\mathbf{D}$ and the reconstructed distances $\hat{\mathbf{D}}$ is maximised by minimising the following embedding cost function:

$$\mathrm{s} = -\mathsf{r} \circ \hat{\mathbf{D}} \circ \hat{\mathbf{X}} \quad \Rightarrow \quad \frac{\partial \mathrm{s}}{\partial \hat{x}_k^i} = -\sum_{j=1\ldots n}^{j\neq i} \frac{\partial \mathsf{r}}{\partial \hat{\mathrm{d}}_{ij}} \cdot \frac{\partial \hat{\mathrm{d}}_{ij}}{\partial \hat{x}_k^i} \to 0,\, i = 1\ldots n \tag{2}$$

Locations of all points $\hat{\mathbf{x}}^i$ in the target space induce pairwise distances and, consequently, correlations between source and target distances. These locations are obtained by gradient descent on the stress function s using the chain rule. The derivatives in Equation 2 are [SSUS07]

$$\begin{aligned}\frac{\partial \mathsf{r}}{\partial \hat{\mathrm{d}}_{ij}} &= \frac{(\mathrm{d}_{ij}-\mu_{\mathbf{D}}) - \frac{\mathscr{B}}{\mathscr{D}}\cdot(\hat{\mathrm{d}}_{ij}-\mu_{\hat{\mathbf{D}}})}{\sqrt{\mathscr{C}\cdot\mathscr{D}}} \\ \frac{\partial \hat{\mathrm{d}}_{ij}}{\partial \hat{x}^i_k} &= (\hat{x}^i_k - \hat{x}^j_k)\,/\,\hat{\mathrm{d}}_{ij} \quad \text{for Euclidean} \quad \hat{\mathrm{d}}_{ij} = \sqrt{\sum\nolimits_{l=1}^{\hat{q}}(\hat{x}^i_l - \hat{x}^j_l)^2}\,.\end{aligned}$$

While for intuitive plotting results target distances $\hat{\mathrm{d}}_{ij}$ are usually Euclidean, input distances can be mere dissimilarities, such as mirrored Pearson correlation $\mathrm{d}_{ij} = (1 - \mathsf{r}(\mathbf{x}^i, \mathbf{x}^j))$ or powers of which. These correlations between data vectors must not be confused with the target value r in the correlation-based cost function optimisation in Equation 2 of HiT-MDS.

Two major revisions are made to the previous version of HiT-MDS described in [SSUS07].

First, the update replaces the specific value of the cost function derivative in Equation 2 by the sign $\mathrm{sgn}(\partial \mathrm{s}/\partial \hat{x}^i_k)$. This forces updates, irrespective of the order of magnitude of the derivative for maintaining a constant convergence process. The effective rate of convergence is controlled by a single factor only, the learning rate γ_t, decreasing in time. Thus, an atomic update quantity of the k-th component of the i-th reconstruction point at time point t is computed by

$$\Delta_t \hat{x}^i_k = -\gamma_t \cdot \mathrm{sgn}\left(\frac{\partial \mathrm{s}}{\partial \hat{x}^i_k}\right), \quad \gamma_t \to 0 \quad \text{for} \quad t \to t_{\max}\,. \tag{3}$$

Convergence is forced by driving the learning rate monotonously to zero, in the limit of maximum cycles $t_{\max}+1$. In practice, the learning rate starts at $\gamma_0 = 0.1$ and gets linearly decreased to zero. This update scheme is very robust against the choice of the learning rate and turns out to yield excellent results.

Secondly, batch optimisation is realised. This means that updates from all pairs of data records are integrated before being applied synchronously to the reconstructed points. This strategy can be formally expressed as operations on distance matrices and, hence, efficiently parallelised. Illustrative MATLAB/Octave and R implementations with vectorised code as well as CUDA codes are available online [Hit].

A general formulation of the point reconstruction procedure is given in Algorithm 1. Much of the work is actually done in line 8 of the program. Apparently, the depicted algorithm is specialised in the task of fast reconstruction of a given dissimilarity matrix $\mathbf{D}$, thereby depending only on the target dimension, adaptation rate, and the number of cycles.

One of the main challenges of transferring the general algorithm to CUDA is an efficient use of memory and threads, which is detailed in the next sections. Another important issue to be discussed is the handling of adjacency matrices for being processed by the HiT-MDS algorithm.

Algorithm 1 General HiT-MDS algorithm

```
1: Initialise $\hat{x}^i_k$ randomly from the unit interval
2: for $t \leftarrow 1 \ldots t_{\max}$ {iterations} do
3:     Calculate distance matrix $\hat{\mathbf{D}}$ of all $\hat{\mathbf{x}}$, including $\mathscr{B}, \mathscr{C}, \mathscr{D}$, and $\mu_{\hat{\mathbf{D}}}$
4:     Calculate update rate $\gamma_t \leftarrow \gamma_0 \cdot (1 - t/(t_{\max} + 1))$
5:     for $k \leftarrow 1 \ldots \hat{q}$ {each target dimension} do
6:         reset $k$-th dimension update vector $\boldsymbol{y} \leftarrow \mathbf{0}$
7:         for $i \leftarrow 1 \ldots n$ {each target point} do
8:             $y_i \leftarrow \Delta_t \hat{x}^i_k$ (using Equation. 3)
9:         end for
10:        for $i \leftarrow 1 \ldots n$ {apply integrated update to each target point} do
11:            $\hat{x}^i_k \leftarrow \hat{x}^i_k + y_i$
12:        end for
13:    end for
14: end for
```

2.2 Implementation on CUDA

As depicted in equations 1, 2 and 3 the essential work of HiT-MDS is done by calculating the Pearson correlation coefficient $\mathsf{r}(\mathbf{D}, \hat{\mathbf{D}})$. Therefore, three intensive summations including mean value computation as well as the Euclidean distances have to be computed. These two tasks are very well suited for being transformed into parallel problems, as described following. We will point out the way of theoretical parallelization complexities and CUDA specific implementation details. Additionally, we included a degree based and a second all-pairs-shortest-path based algorithm to enhance the graph distance interpretation possibilities of HiT-MDS for creating network layouts.

All pairwise distances are stored in a half n^2-matrix and accessed by a function $getPos(x, y) = row\_coord[y] + x$ for a graph $G = (V, E)$, $(x, y) \in E$. row_coord contains pre-calculated coordinates of starting points for each row of the given half matrix. On the GPU this is realised as an array in fast constant memory to provide best access times.

The Prefix Reduction or partial-sums problem is a well understood algorithmic approach to maintain partial sums of a given array $A[1..n]$. It is specified for elements $A[i]$ to come from an arbitrary group H containing at least 2^δ elements. For the cell-probe model with b-bit cells a problem complexity of $\Omega = \left(\frac{\delta}{b} \cdot \lg n\right)$ was proven [PD04].

For parallel implementations, it was shown that the naïve algorithm's time complexity is $\mathcal{O}(\log n)$ performing $\mathcal{O}(n \log_2 n)$ addition operations. Furthermore, work efficient implementations (Algorithm 2) perform only $\mathcal{O}(n)$ addition operations [HSO07].

To get additional speed improvements, it is necessary to take advantage of the multiprocessors shared memory. All threads running in the same block have communication access to the same shared memory. In this case, communication means to copy all elements known

Algorithm 2 Work-efficient partial-sum algorithm

```
1: for $d \leftarrow 0 \ldots \log_2 n - 1$ do
2:     for $k \leftarrow 0 \ldots n - 1$ by $2^{d+1}$ in parallel do
3:         $A[k + 2^{d+1} - 1] \leftarrow A[k + 2^d - 1] + A[k + 2^{d+1} - 1]$
4:     end for
5: end for
```

by a thread into a smaller shared memory array. As mentioned above, shared memory is substantially faster than global memory.

To avoid bank conflicts, we use the blockIDs and threadIDs to calculate memory addresses of elements in shared memory that a thread adds up, schematically given as

$$sum = shared\,[threadID] + shared\,[threadID + blockDim/2] \tag{4}$$

Thus, with a limitation to the maximum number of threads per block of 512, caused by CUDA constraints, one can add 1024 elements per block. The result is written back to global memory and is source of the next loop. Therefore, in every step of outer for-loop of Algorithm 2, there are $2 \cdot k$ global memory accesses. To take advance of coalescing memory, we address the elements read from global memory by a similar idea as in formula 4.

Euclidean Distances computation time complexity is in $\mathcal{O}\left(\frac{n^2}{p}\right)$. We reached acceleration factors of more than 20 up to 30 by using a simple kernel according to Algorithm 3. This approach uses the CUDA built-in register variables $id.x$ and $id.y$ to find out the virtual location of an active thread.

Algorithm 3 Parallel pairwise Euclidean distances with thread IDs in $\hat{q}$-dimensional space

```
1: for $i \leftarrow 0 \ldots n^2$ in parallel do
2:     $\hat{\mathrm{d}}_{id.x,id.y} \leftarrow \sqrt{\sum_{l=1}^{\hat{q}} (\hat{x}_l^{id.x} - \hat{x}_l^{id.y})^2}$
3: end for
```

Floyd-Warshall Algorithm According to use MDS as graph layout tool, a simple approach to get more information out of sparse graphs is to compute extra distances from existing graph edges by finding all node pairs shortest path. The Floyd-Warshall algorithm was designed with this in mind and is, similar to the Euclidean distances algorithm, very simple to transform into a parallel version. It is a single $\mathcal{O}(n)$ operation looping over $\mathcal{O}(n)$ threads as shown in Algorithm 4 and pointed out by Harish and Narayanan [HN07] who reported significant speed improvements. Again, we use the texture memory access method to profit from caching effects.

Degree Based Distance Manipulation An alternative and fast method to visualise network structures is to pre-compute distances out of adjacencies. The main idea is to declare

Algorithm 4 Parallel Floyd-Warshall(G=(V,E))

```
1: create adjacency matrix $A$ from $G$
2: for $k \leftarrow 1 \ldots n$ do
3:     for all elements in the adjacency matrix $A$, where $1 \leq i, j \leq n$ in parallel do
4:         $A[i, j] \leftarrow \min(A[i, j], A[i, k] + A[k, i])$
5:     end for
6: end for
```

nodes with many neighbours as large nodes. Basically, this method is in a time complexity of $\mathcal{O}(no)$ with o is the average number of neighbours per node. Since most networks in biology are sparse, the algorithm works very fast. In substance, in such a network $G = (V, E)$ the distances are defined as $l(e) = \deg(u) + \deg(v)$ for $e = (u, v) \in E$.

HiT-MDS is applied two times on the pre-computed distance values using half of the standard cycle number each. In the first run, we set unknown distances belonging to different components to the graph's doubled diameter. This first step separates all components. In the second step, and hence, during the second half of total algorithm cycles, points are moved to their best correlation based positions. A result of this approach is given in Figure 2.

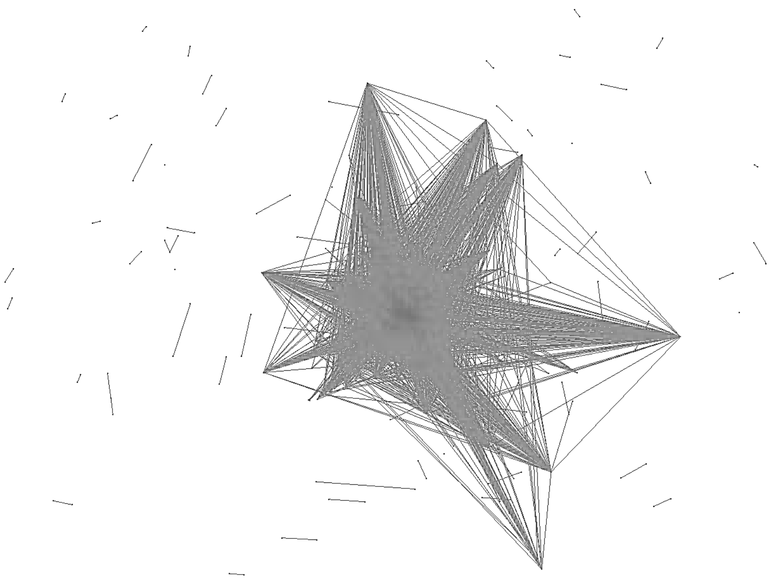

Figure 2: Yeast protein interaction network with 4554 nodes, evaluated with degree based distance interpretation. The node positions are visualised on the basis of CUDA's OpenGL-interoperability.

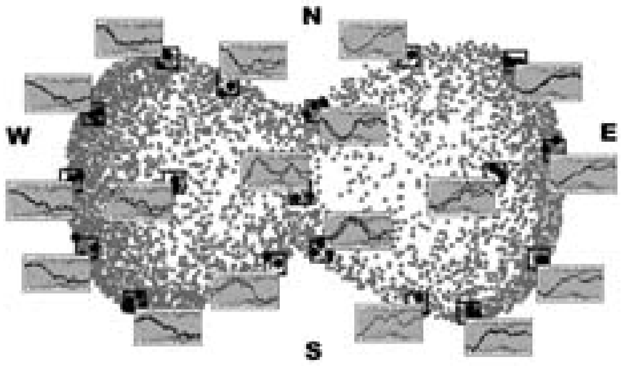

Figure 3: HiT-MDS scatter plot of embedded temporal gene expression data. Correlation similarity $(1 - \mathsf{r}(\mathbf{x}^i, \mathbf{x}^j))^p$ is considered at $p = 8$ for magnification of high-correlation subsets, which explains the characteristic sand glass shape.

2.3 Application Examples

2.3.1 Global Patterns of Gene Expression

Visualisation is sought for 4824 high-quality genes covering 14 time points of developing Barley
grains [SSUS07]. Their scatter plot is obtained by running HiT-MDS for 50 data cycles, yielding the high-quality display shown in Figure 3. In contrast to previous results the processing time dropped from 861 seconds by a sequential C program to merely 6 seconds using CUDA (not including disk read).

The characteristic sand glass shape results from using eighth power of the correlation measure, more precisely, $(1 - \mathsf{r}(\mathbf{x}^i, \mathbf{x}^j))^8$, applied to highly correlated 14-dimensional time series profiles of up-regulation vs. down-regulation processes. The power of eight magnifies subtle dissimilarities in highly correlated genes, this way enhancing their visual differentiation. By posterior labelling with known gene annotations, the exemplary group of hormone and signaling related genes are highlighted in orange colours, other functional categories are marked in gray. Additionally, data boxes, brushed in blue, and their corresponding plots of temporal patterns have been manually picked in order to demonstrate the high spatial connectivity of similar regulatory profiles and their embedded two-dimensional counterparts. A smooth transition can be found from the western side (W) with patterns of down-regulation, via south (S) corresponding to patterns of intermediate up-regulation and up-regulation located in the east (E) to north (N) with intermediate down-regulation, back to west. Since the underlying array was designed for capturing gene expression connected to developmental processes, the majority of genes is in fact expected to be either up- or down-regulated, as visually confirmed by the two major structures. Rare and unique regulation patterns are found in the interior of the sand glass shape.

The prominent temporal expression patterns are easily revealed by browsing the scatter plot in the way described above. The plot shows that the correlation space is very homogeneous, dominated by patterns of up- and down-regulation, according to the experimental design. Overall, the HiT-MDS embedding procedure applied to transcriptome data of barley tissue development yields a faithful arrangement of genes with their typical temporal expressions. Together with functional annotation data this is a very instrumental tool for screening sets of co-regulation and for an initial derivation of tentative pathways.

2.3.2 Network Layout

Many processes and interactions in biology are represented as networks. Furthermore, there are two common ways to interpret experimental data resulting in networks: i) as a biological network and ii) in the context of an underlying network. Due to the increasing amount of experimental data and the steadily growing size of networks, automatic network layout is important to better understand the relationships and interactions between biological objects such as genes, transcripts, proteins and metabolites. One widely used method for network layout is the force-directed layout method [FR91].

Let $G = (V, E)$ be a network consisting of a set of nodes $V = \{v_1, \dots, v_n\}$ representing the biological objects (e. g. proteins) and a set of edges $E = \{(v_i, v_j) | v_i, v_j \in V\}$ representing the interactions between the biological objects (e. g. interactions between proteins). A layout of the network is represented by coordinates for the nodes and curves for the edges. A force-directed layout method uses a physical analogy to draw networks. It simulates a system of physical forces defined on the network and produces a drawing which represents a locally minimal energy configuration of that physical system. Force-directed layout methods consist of two parts: i) a system of forces defined by the nodes and edges, and ii) a method to find positions for the nodes (representing the layout of the network) such that for each node the total force is (close to) zero.

A typical method interprets nodes as mutually repulsive 'particles' and edges as 'springs' connecting these particles. This results in attractive forces f_a between adjacent nodes and repulsive forces f_r between non-adjacent nodes. For the current layout for each node $v \in V$ the force $F(v) = \sum_{(u,v) \in E} f_a(u, v) + \sum_{(u,v) \in V \times V} f_r(u, v)$ is computed, which is the sum of all attractive forces f_a and all repulsive forces f_r affecting node v. For example, for the x component the forces f_a and f_r are defined as $f_a(u, v) = c_1 \cdot (d(u, v) - l) \cdot \frac{x(v) - x(u)}{d(u,v)}$ and $f_r(u, v) = \frac{c_2}{d(u,v)^2} \cdot \frac{x(v) - x(u)}{d(u,v)}$, respectively, where l is the optimal distance between any pair of adjacent nodes, $d(u, v)$ is the current distance between the nodes u and v, $x(v)$ is the x-coordinate of node v, and c_1, c_2 are positive constants. Iterative numerical analysis is used to find a locally minimal energy configuration by moving each node in the direction of $F(v)$ to produce a new layout. Finally, the nodes are connected by straight lines.

There are several often used varieties of force-directed methods [Ead84, KK89, SM95]. The computation of the layout is computationally demanding, and fast force-directed methods have been proposed such as an incremental multidimensional scaling heuristic [Bas99] or Walshaw's algorithm (a multi level version of the original algorithm [FR91]) in [HJP02]. The previously shown network example in Figure 2 is based on the HiT-MDS node layout for visualizing a protein interaction network in yeast containing 4554 nodes.

instant size	**MATLAB** [s]	**CUDA** [s]	**speedup**
64	0.114±0.010	0.012±0.004	9.5 x
128	0.126±0.006	0.014±0.007	9.0 x
256	0.616±0.012	0.021±0.003	29.3 x
512	2.777±0.205	0.048±0.004	57.9 x
1024	9.975±0.485	0.178±0.004	56.0 x
2048	43.473±2.832	0.721±0.003	60.3 x
4096	183.700±11.915	3.361±0.024	54.7 x
8192	750.129±48.643	13.997±0.015	53.6 x

Table 1: Performance comparison of optimized MATLAB and CUDA code on test instances of different sizes. The target dimension is three. Measurements refer to time in seconds, excluding data import time.

3 Results

The core HiT-MDS algorithm has been implemented on three different platforms. Two vectorized code samples are available for R and MATLAB/GNU Octave as well as the CUDA version. Code profiling tools of MATLAB and CUDA were used to optimize the performance. Code for R and GNU Octave was manually optimized and is by a factor of 2 to 4 slower than the MATLAB version, because only MATLAB is able to make use of fast single precision arithmetics, thereby using multi-threaded linear algebra routines and loop optimization. Therefore, the fastest code of MATLAB is compared with the CUDA implementation.

Random distance matrices of different sizes were generated for performance tests on the reference server machine, a 16 core server equipped with 3 GHz AMD Opteron CPUs and a NVidia TESLA S870 GPU rack. MATLAB 7.7.0 with multi-thread mode and CUDA 2.1 were used for performance comparisons. Average run times of 10 independent starts with 50 cycles per run were measured and compiled in Table 1. The instant size column refers to matrices representing between 64x64 to 8192x8192 distances. For the fixed number of cycles, the embedding speed is independent of the matrix entries, no matter if full or sparse matrices are processed. This also indicates a general validity of the recorded speed, no matter if for scatter plot generation or for network layout.

Significantly faster execution times of CUDA are found. Yet, small instances yield less speedup than instances of sizes around 2048x2048 for which robust factors over 50 fold acceleration can be stated. Moreover, very small standard deviations are obtained for CUDA, indicating undisturbed use of the GPU hardware for high-performance scientific calculations.

4 Discussion

HiT-MDS is a versatile algorithm with good parallelization potential for reconstruction of dissimilarity relationships in a Euclidean space. It can be used as faithful dimension reduction method, for converting data with a specific data metric into a Euclidean representation, and, by a straight-forward extension, for network reconstruction of adjacency matrices. The method is thus perfectly suited for dealing with data screening, complexity reduction, and relationship characterization, tasks that regularly exist in biological sciences.

At first glance the presented performance comparison between MATLAB and CUDA might seem to be unfair. Yet, only few lines of MATLAB code need to be interpreted per algorithmic cycle. Virtually all matrix operations are handled by internal MATLAB functions of optimized algebra subroutines that can be hardly beaten by hand-written C++ code. Just another theoretical factor of 2 would be gained for MATLAB if symmetry of the matrices could be efficiently exploited. Yet another clear time benefit of CUDA remains: thanks to the GPU server architecture heavy computations can run almost independent of the host, if memory transfer between CPU and GPU remains at a low level.

The main ingredients to successful utilization of CUDA turned out to be (i) the consistent use of the reduction principle for using fast shared memory on the multiprocessors instead of slow global memory on the graphics board, (ii) the use of texture memory, and (iii) a good arrangement of threads into logical blocks, and (vi) the use of thread-interleaving memory access (coalescence). For the computing task at hand, single precision floating point numbers of 32 bit worked as reliably as double precision. The technical limitation of the size of the distance matrix is currently at about 14000x14000 elements on a graphics board with 1.5 GB memory. Yet, larger memory capacity, double precision calculations, and more multiprocessors per GPU are already available at low prices.

Future tasks are related to deal with larger network structures, which requires an implementation of a sparse matrix data structure. Another challenge will be the identification of network nodes in very large graph structures.

References

[Bas99] W. Basalaj. Incremental Multidimensional Scaling Method for Database Visualization. In R. F. Erbacher, P. C. Chen, and C. M. Wittenbrink, editors, *Visual Data Exploration and Analysis VI (Proc. SPIE)*, volume 3643 of *Proceedings of SPIE*, pages 149–158, 1999.

[BK09] J. Breitbart and G. Khanna. An exploration of CUDA and CBEA for a gravitational wave data-analysis application. Einstein@Home, 2009.

[CMS08] S. Che, J. Meng, and J. W. Sheaffer. A performance study of general purpose applications on graphics processors using CUDA. *Journal of Parallel and Distributed Computing*, 68(10):1370–1380, 2008.

[Cor08] NVIDIA Corporation. *NVIDIA CUDA Programming Guide Version 2.1*, 12/8/2008.

[Ead84] P. Eades. A Heuristic for Graph Drawing. *Congressus Numerantium*, 42:149–160, 1984.

[FJ07] M. Fatica and W. Jeong. *Accelerating MATLAB with CUDA*. HPEC, 2007.

[FR91] T. Fruchterman and E. Reingold. Graph Drawing by Force-directed Placement. *Software - Practice and Experience*, 21(11):1129–1164, 1991.

[GDD08] N. A. Gumerov, R. Duraiswami, and W. Dorland. *Efficient Personal Supercomputing in Fortran 9x on CPU-GPU Systems*. CSCAMM, 2008.

[GHGC09] A. Godiyal, J. Hoberock, M. Garland, and J. C.Hart. Rapid Multipole Graph Drawing on the GPU. In *Graph Drawing*, volume 5417 of *LNCS*, pages 90–101. Springer, 2009.

[Hit] HiT-MDS at IPK Gatersleben - Data Inspection Group. http://dig.ipk-gatersleben.de/.

[HJP02] K. Han, B.-H. Ju, and J. H. Park. InterViewer: Dynamic Visualization of Protein-Protein Interactions. In M. T. Goodrich and S. G. Kobourov, editors, *Graph Drawing (Proc. GD '02)*, volume 2528 of *LNCS*, pages 364–365. Springer, 2002.

[HN07] P. Harish and P.J. Narayanan. Accelerating Large Graph Algorithms on the GPU Using CUDA. In *High Performance Computing*, volume 4873 of *LNCS*, pages 197–208. Springer, 2007.

[HSO07] M. Harris, S. Sengupta, and J. D. Owens. Parallel Prefix Sum (Scan) with CUDA. In *GPU Gems 3*, pages 851–875. NVIDIA Corporation, 2007.

[IMO09] S. Ingram, T. Munzner, and M. Olano. Glimmer: Multilevel MDS on the GPU. *IEEE Transactions on Visualization and Computer Graphics*, 15(2):249–261, 2009.

[JK09] M. Januszewski and M. Kostur. Accelerating numerical solution of Stochastic Differential Equations with CUDA. *ArXiv e-prints*, 2009.

[KK89] T. Kamada and S. Kawai. An Algorithm for Drawing General Undirected Graphs. *Information Processing Letters*, 31(1):7–15, 1989.

[LKPM09] H. Li, A. Kolpas, L. Petzold, and J. Moehlis. Parallel Simulation for a Fish Schooling Model on a General-Purpose Graphics Processing Unit. In *Concurrency and Computation: Practice and Experience*, 21(6), pages 725–737, 2009.

[PD04] M. Pătraşcu and Demaine. Lower Bounds for Dynamic Connectivity. In *Encyclopedia of Algorithms*, pages 473–477. Springer, 2004.

[RRS$^+$07] S. Ryoo, C. I. Rodrigues, S. S. Stone, S. S. Baghsorkhi, S. Ueng, and W. W. Hwu. Program optimization study on a 128-core GPU. Presented at the First Workshop on General Purpose Processing on Graphics Processing Units, October 2007.

[Sel08] J. Seland. CUDA Programming, 2008. http://heim.ifi.uio.no/ knutm/geilo2008/seland.pdf.

[SM95] K. Sugiyama and K. Misue. A Simple and Unified Method for Drawing Graphs: Magnetic-Spring Algorithm. In R. Tamassia and I. G. Tollis, editors, *Graph Drawing (Proc. GD'94)*, volume 894 of *LNCS*, pages 364–375. Springer, 1995.

[SSUS07] M. Strickert, N. Sreenivasulu, B. Usadel, and U. Seiffert. Correlation-maximizing surrogate gene space for visual mining of gene expression patterns in developing barley endosperm tissue. *BMC Bioinformatics*, 8:165, 2007.

[TO05] Y.-H. Taguchi and Y. Oono. Relational patterns of gene expression via non-metric multidimensional scaling analysis. *Bioinformatics*, 21(6):730–740, 2005.